Hands-On 7
Quasi Monte Carlo Methods

In this *Hands On* series we will demonstrate how to apply Quasi Monte Carlo (QMC) techniques to compute our favorite integrals/expectations, in order to numerical assess their and compare them with crude Monte Carlo estimates. As already done, we will consider a simple test function throughout this section, instead of taking more complicated examples (e.g., where one function evaluation could involve solving a differential problem). The exposition is intentionally step by step. **This time no exercises, only fun. :-)**

We consider the following function

$$g(\mathbf{x}) = \exp\left(c \sum_{j=1}^{d} x_j j^{-b}\right) = \prod_{j=1}^{d} \exp(c x_j j^{-b})$$

for which, for testing purposes, we know that

$$I(g) = \int_{[0,1]^d} g(\mathbf{x}) d\mathbf{x} = \prod_{j=1}^{d} \frac{\exp(cj^{-b}) - 1}{cj^{-b}}, \qquad c \neq 0.$$

In particular, we are interested to *large scale* problems, for which the dimension d becomes large; we will fix, for instance, $d = 100$.

This material is substantially taken from `https://people.cs.kuleuven.be/~dirk.nuyens/taiwan/`, where you can also find notes and Matlab codes.

1 Setting

Let us define g in Matlab as a vectorized inline function taking multiple vectors at once as an $d \times n$ array and returning a $1 \times n$ array of results:

```
% x is an [s-by-n] matrix, n vectors of size s; vectorized g(x):
g = @(x, c, b) exp(c * (1:size(x,1)).^(-b) * x);
% note that vectors are considered to be columns and so the
% product above is an inner product, summing over the dimensions
```

In fact we will define g in a slightly different way as we will not just pass in multiple vectors at once, but also different shifted versions of these vectors as an $d \times n \times m$ array:

```
g = @(x, c, b) reshape( ...
   exp(c * (1:size(x,1)).^(-b) * x(:,:)), ... % 'as above'
   1, size(x,2), size(x,3) ... % 1-by-n-by-[whatever left]
);
```

Of course more complicated functions would better be defined in a separate file. For instance, we could define g in a separate file with the same function signature (we call this file `gfun.m`):

```
function y = gfun(x, c, b)
% function y = gfun(x, c, b)
%
% Vectorized evaluation of the example function
% \[ g(x) := \exp( c \sum_{j=1}^s x_j j^{-b} ) \]
%
```

```
% Inputs:
%   x     array of s-dimensional vectors, [s-by-n] array
%          or [s-by-n-by-m] array (or even deeper nesting,
%          but the first dimension should be the dimension s)
%   c     scaling parameter, scalar
%   b     dimension decay parameter to diminish influence of
%          higher dimensions, scalar, b >= 0 (b = 0 is no decay)
%
%  Outputs:
%   y     function value for each input vector, [1-by-n] array
%          or [d-by-n-by-m] array (or deeper, but same as x)
%
% Note: the array x (and also the resulting array y) can have more
% than two dimensions, e.g., x could be [s-by-n-by-m] and then the
% resulting y will be [1-by-n-by-m]. This is to accommodate for
% multiple versions of a point set (e.g., for shifted point sets).

y = reshape( ...
        exp(c * (1:size(x,1)).^(-b) * x(:,:)), ...
        max((1:ndims(x) ~= 1) .* size(x), 1) ... % first dim to 1
    );
```

This version is even more general as it allows x to have any shape as long as the leading dimension is d (the dimensionality of the points), and it will return an array of the same shape but with the first dimension set to 1 (mapping vectors into function values). The way we have chosen to lay out our collections of d-dimensional vectors is such that the vectors are stored consecutively in memory. In this way the data needed to evaluate one function value is localized in memory.

We are now ready to fix some parameters:

```
% parameters of the g-function
d = 100;  % number of dimensions
c = 1;    % c-parameter
b = 2;    % decay-parameter
```

Then we can calculate its exact integral value:

```
a = c * (1:d).^(-b); exact = prod(expm1(a)./a);
% or as a function (repeating the 'a' twice):

gexact = @(s, c, b) prod(expm1(c*(1:d).^(-b))./(c*(1:d).^(-b)));
% notice we use expm1(a) and not exp(a) - 1
```

Note the usage of the function `expm1` in calculating the exact integral of g: this is useful for small arguments when $\exp(x) \approx 1$. The parameters c and b specify how difficult the function g will be. We can use Figure 1 as a guideline. It is clear we have a product of such one-dimensional exponential functions.

Looking at the figure, we see that if the argument to the exponential function is a small number then the function is nearly linear and approaching a constant function. Note that constant functions are ridiculously easy to integrate. The extreme case would be $c = 0$; in that case both MC and QMC methods will give the exact value of the integral already with one function value.

The larger the value of c (positive or negative) the more we deviate from a linear function and we need more and more samples to determine its integral. For negative c with very large magnitude, the function is essentially 0 except for $g(0) = 1$, so it is a "peaky function" and rather hard to integrate.

In the multivariate case, the parameter b, with $b \geq 0$, modifies how quickly we converge to a constant function as the dimension j increases. When the number of dimensions increases, the deviation from the constant function is "multiplied".

For the sake of reproducibilty, we will use the Mersenne Twister as the random number generator for our MC simulations and we will set its initial state to a fixed value such that we can repeat our experiment and get exactly the same random numbers. Similarly, we will use the combined recursive generator from L'Ecuyer for the random shifting in case of the QMC approximations. This is also a possible way to see some ideas explored in Chapter 1 in action.

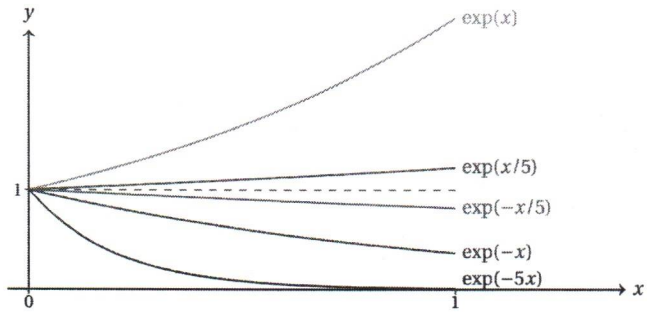

Figure 1: Interpretation of the combined parameters c and b for the function g in a single coordinate. The effect is "multiplied" in multiple dimensions.

```
ng_MC = RandStream('mt19937ar', 'Seed', 1234);
rng_shifts = RandStream('mrg32k3a', 'Seed', 1234);
```

In Matlab we can now draw random numbers from the Mersenne Twister by doing `x = rand(rng_MC, d, n)` to obtain a $d \times n$ array.

2 Monte Carlo approximation

We are now ready to do a first approximation of the integral using CMC method. We will use hundred thousands samples to get a MC approximation. (Of course in our test case we do know the exact value of the integral.)

```
tic;
N = 1e5; % number of samples
G = g(rand(rng_MC, d, N), c, b); % evaluate at once, mean but easy
MC_Q = mean(G);
MC_std = std(G)/sqrt(N);
t = toc;
fprintf('MC_Q = %g (error=%g, std=%g, N=%d) in %f sec\n', ...
   MC_Q, abs(MC_Q - gexact(d, c, b)), MC_std, N, t);
```

This gives us

```
MC_Q = 2.3679 (error=0.000573539, std=0.00223394, N=100000) in 0.183208 sec
```

Without resetting the seed and re-running the above code 9 times gives the following output:

```
MC_Q = 2.36737 (error=0.00110043, std=0.00223924, N=100000) in 0.095873 sec
MC_Q = 2.37118 (error=0.00270455, std=0.00224065, N=100000) in 0.096617 sec
MC_Q = 2.36787 (error=0.00060052, std=0.00224386, N=100000) in 0.094095 sec
MC_Q = 2.36933 (error=0.00086130, std=0.00223436, N=100000) in 0.097445 sec
MC_Q = 2.37172 (error=0.00324287, std=0.00223811, N=100000) in 0.094814 sec
MC_Q = 2.37063 (error=0.00216023, std=0.00223951, N=100000) in 0.094031 sec
MC_Q = 2.36794 (error=0.00053736, std=0.00224293, N=100000) in 0.092464 sec
MC_Q = 2.36981 (error=0.00133505, std=0.00223974, N=100000) in 0.094705 sec
MC_Q = 2.36995 (error=0.00147575, std=0.00223747, N=100000) in 0.095543 sec
```

In Figure 2 we plot the results of 5 approximations to obtain estimates to $I(g)$ as well as the standard errors plotted in terms of the number of samples used. We can clearly see from the figure that the convergence is $1/\sqrt{N}$, as expected. Using $5 \cdot 10^4$ random samples we observe an estimated standard error of $\approx 10^{-3}$. If we would want to divide this error by 10 then we will need to take 100 times more samples.

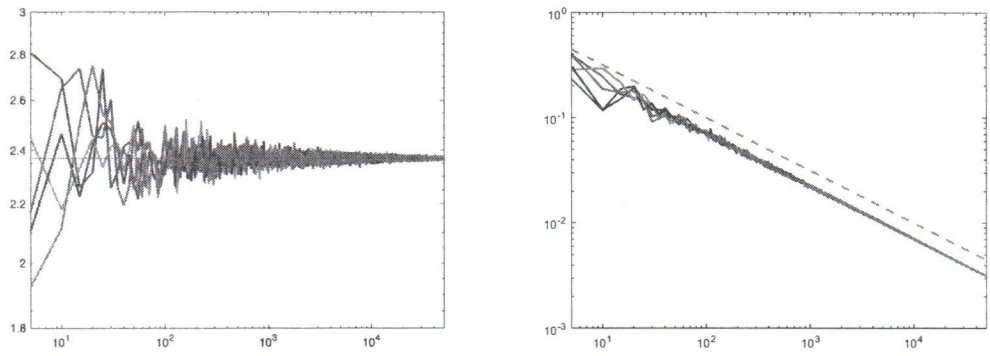

Figure 2: Left: Monte Carlo integral approximation for 5 runs for the function g with $c = 1, b = 2$ and $d = 100$. Right: Standard error of 5 Monte Carlo integral approximations for g with $c = 1, b = 2$ and $d = 100$. The straight line is $1/\sqrt{N}$.

3 Quasi Monte Carlo approximation

As we just observed the MC rate of $1/\sqrt{N}$ we are ready to try a QMC rule and see if we get the promised $1/\mathrm{N}$ convergence. We take the simplest of all QMC rules: a lattice rule (or actually a lattice sequence).

We take the simplest of all QMC rules: a lattice rule (or actually a lattice sequence). We will use a "multi purpose" lattice sequence to calculate our integral. We use the generating vector from the file

`exod2_base2_m20_CKN`

on the webpage `https://people.cs.kuleuven.be/~dirk.nuyens/qmc-generators/`. The maximum number of dimensions provided is 250 and the maximum number of points is 2^{20} (about a million). The rule was constructed to be good for all powers of 2 and can thus be used to obtain a sequence of approximations.

```
z = load('exod2_base2_m20_CKN.txt'); % should be column vector!
z = z(:); % will force column vector
tic;
N = pow2(16); x = mod(z(1:d) * (0:N-1), N) / N; % all 2^16 points
QMC_Q = mean(g(x, c, b));
t = toc;
fprintf('QMC_Q = %g (error=%g, N=%d) in %g sec\n', ...
        QMC_Q, abs(QMC_Q - gexact(d, c, b)), N, t);
```

This gives us

```
QMC_Q = 2.36845 (error=2.10821e-05, N=65536) in 0.176885 sec
```

As this is a deterministic result, there is no standard error output on the estimator. In this case of $c = 1$ we get a nice error of $\approx 10^{-5}$ while MC only gets $\approx 10^{-3}$ using 10^5 evaluations, i.e., more than 50% more evaluations. Although the complexity of generating random numbers and generating lattice points is technically almost the same, there is a larger difference in execution time in `Matlab` due to the fact that the `Matlab` random number generator is compiled code; we already saw however that the error of QMC is much much smaller than MC for the same number of points. To obtain a practical error estimate we will now use several randomly shifted lattice rules and look at successive approximations for increasing number of points.

```
M = pow2(3); % number of random shifts
shifts = rand(rng_shifts, d, M);

Ns = pow2(4:16);              % calculate approximations for different N's
QMC_Q = zeros(size(Ns));      % approximations for increasing N
QMC_std = zeros(size(Ns));    % standard error for each N
QMC_VG = zeros(size(Ns));     % estimate of variance of g for each N
```

```
for i=1:numel(Ns)

   N = Ns(i); % total number of function evaluations
   n = N / M; % use QMC rule with n = (N/M) points and
              % M independent realizations of this rule

   tic; % we generate all points and all shifts d-by-n-by-M:
   G = g(mod(bsxfun(@plus, reshape(z(1:d)*(0:n-1), [d n 1]), ...
      reshape(n*shifts, [d 1 M])), n)/n, ...
         c, b);      % G is 1-by-n-by-M (all points, all shifts)
   QMC_R        = mean(G);                    % mean(G) is 1-by-1-by-M
   QMC_Q(i)     = mean(QMC_R);          % average over all M shifts
   QMC_std(i)   = std(QMC_R)/sqrt(M);    % stderr over M shifts
   t = toc;

   % ad hoc calculations of var(g)
   QMC_VG(i) = sum(G(:).^2)/N - mean(G(:)).^2;
   fprintf('QMC_Q = %g (error=%g, std=%g, N=%d) in %f sec\n', ...
      QMC_Q(i), abs(QMC_Q(i) - gexact(d, c, b)), ...
      QMC_std(i), N, t);
end
```

This gives us

```
QMC_Q = 2.39186 (error=0.0233848, std=0.110352, N=16) in 0.001852 sec
QMC_Q = 2.38799 (error=0.0195154, std=0.0491337, N=32) in 0.000579 sec
QMC_Q = 2.3527 (error=0.0157772, std=0.0246321, N=64) in 0.000921 sec
QMC_Q = 2.34172 (error=0.0267508, std=0.0107814, N=128) in 0.002096 sec
QMC_Q = 2.35169 (error=0.0167805, std=0.00589039, N=256) in 0.005889 sec
QMC_Q = 2.37198 (error=0.00350746, std=0.00513181, N=512) in 0.003398 sec
QMC_Q = 2.36877 (error=0.000294541, std=0.00225967, N=1024) in 0.005175 sec
QMC_Q = 2.36647 (error=0.00200395, std=0.000959644, N=2048) in 0.009424 sec
QMC_Q = 2.3671 (error=0.00137409, std=0.000555156, N=4096) in 0.020618 sec
QMC_Q = 2.36797 (error=0.000499672, std=0.000190588, N=8192) in 0.039067 sec
QMC_Q = 2.36828 (error=0.000188277, std=0.000116317, N=16384) in 0.051808 sec
QMC_Q = 2.36857 (error=9.36751e-05, std=6.72594e-05, N=32768) in 0.091855 sec
QMC_Q = 2.36853 (error=6.12873e-05, std=3.90285e-05, N=65536) in 0.184394 sec
```

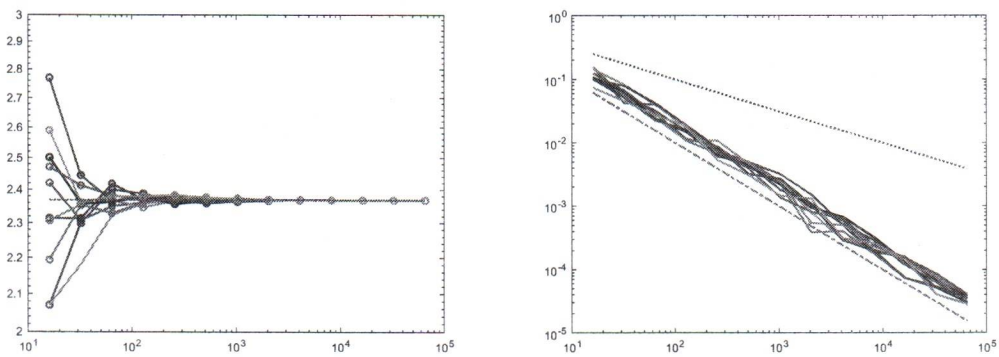

Figure 3: Left: Quasi Monte Carlo integral approximation for randomly shifted lattice rules (with 8 shifts), for the function g with $c = 1, b = 2$ and $s = 100$. We plot 10 results (for 10 times 8 random shifts) to get an idea of how the result varies in terms of the randomness of the shifts. Right: Standard error of 10 quasi-Monte Carlo integral approximations. Superimposed, the rates $N^{-1/2}$ and N are reported.

The results of these QMC simulations with different shifts are given in Figure 3. The difference with the MC simulations (greyed out on the plots) is enormous, which is most obvious on the log-log plot of the standard error. The convergence rate here is near $1/N$. If we want to have one more digit accuracy, then we need to take 10 times more samples (instead of the 100 in the case of MC). This is the power of plain straightforward QMC.

Below, in Figure 4 we compare the estimates of the integral (with increasing N) and the standard deviation, using the same number of points $2^4, \ldots, 2^{16}$ for both MC and QMC formulas. A decrease in variance with the QMC formula is also clearly visible.

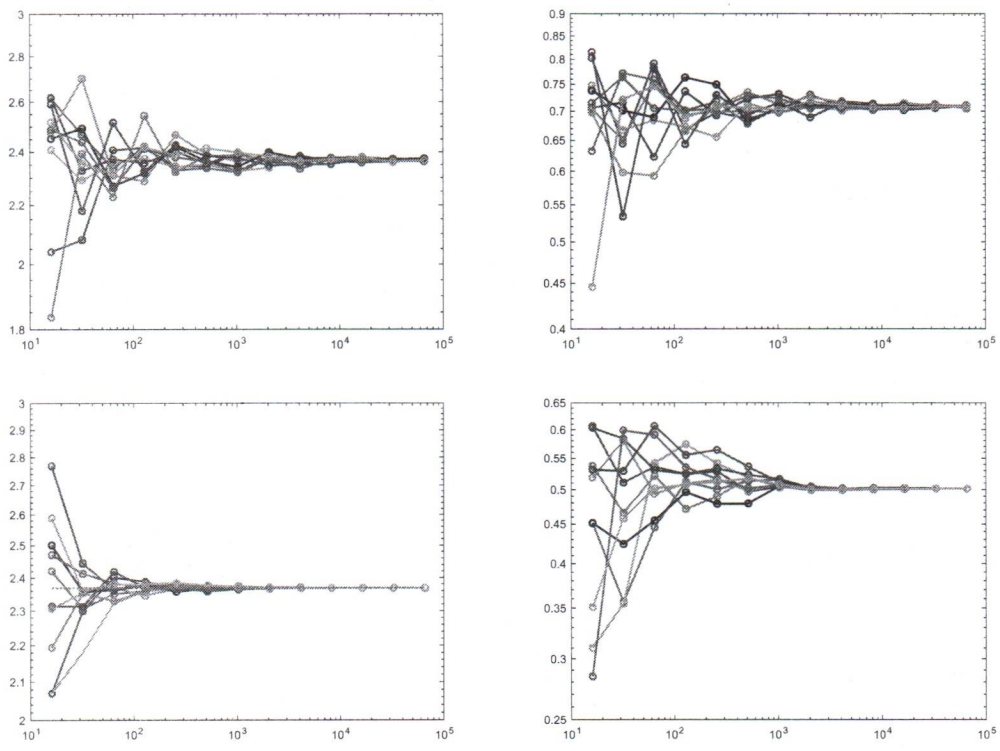

Figure 4: Top: Monte Carlo integral approximation for randomly selected points, 10 runs. Bottom: Quasi Monte Carlo integral approximation for randomly shifted lattice rules (with 8 shifts), 10 runs (for 10 times 8 random shifts). Left: MC or QMC estimates. Right: standard deviations.